

Modularizing and Specifying Protocols among Threads

Sung-Shik T.Q. Jongmans

Centrum Wiskunde & Informatica (CWI)
Amsterdam, the Netherlands
jongmans@cwi.nl

Farhad Arbab

Centrum Wiskunde & Informatica (CWI)
Amsterdam, the Netherlands
farhad.arbab@cwi.nl

We identify three problems with current techniques for implementing protocols among threads, which complicate and impair the scalability of multicore software development: implementing synchronization, implementing coordination, and modularizing protocols. To mend these deficiencies, we argue for the use of domain-specific languages (DSL) based on existing models of concurrency. To demonstrate the feasibility of this proposal, we explain how to use the model of concurrency Reo as a high-level protocol DSL, which offers appropriate abstractions and a natural separation of protocols and computations. We describe a Reo-to-Java compiler and illustrate its use through examples.

1 Introduction

With the advent of multicore processors, a new era began for many software developers of general, non-numerical, applications: to harness the power of multicore processors, the need for writing concurrently executable code, instead of traditional sequential programs, intensified—a notoriously difficult task with the currently popular tools and technology! To alleviate the burden of implementing concurrent applications, researchers started developing new techniques for multicore programming. Examples include stream processing, transactional memory, and lock-free synchronization. However, one rather high-level aspect of multicore programming has received only little attention: the sets of rules that interacting parties must abide by when they communicate with each other—*protocols*. In this paper, we investigate a new approach for implementing protocols among threads.

Many popular general-purpose programming languages (GPPL) feature threads: concurrently executing program fragments sharing the same address space. To name a few such GPPLs and (some of) their multithreading facilities:

- Fortran has coarrays and OpenMP;
- C and C++ have Pthreads and OpenMP;
- Objective-C has Pthreads and the NSThread class;
- Visual Basic and C# have the `System.Threading` namespace;
- Java has the `Thread` class.

These languages have a combined share of roughly 63% according to the TLP and Tiobe indexes of January 2012.^{1,2} From these statistics, one can conclude that (a good portion of) sixty percent of software developers encounters threads regularly. Consequently, many developers will benefit from improvements to existing techniques for implementing protocols among threads. We consider such improvements not merely relevant but a sheer necessity: the current models and languages, APIs and libraries, fail to scale

¹ <http://lang-index.sourceforge.net/>

² <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

when it comes to implementing protocols. Importantly, we refer to scalability not only in terms of performance but also in terms of other aspects of software development (e.g., correctness, maintainability, and reusability). Our approach in this paper takes such aspects into account.

Organization In Section 2, we identify three problems with current techniques for implementing protocols among threads. In Section 3, we sketch an abstract solution based on the general notion of domain-specific languages. In Section 4, we concretize our approach for one particular such language, namely Reo. Section 5 concludes this paper.

2 Problems with Implementing Protocols

While threads prevail for implementing concurrency in general-purpose programming languages (GPPL), they also provoke controversy. Programming with threads would inflict unreasonable demands on the reasoning capabilities of software developers, partly due to the unpredictable ways in which threads interact with each other [7]: typically, one cannot analyze all the ways in which threads may interleave, and consequently, unforeseen—and potentially dangerous—execution paths may exist. Some propose to discard our present notion of threads, unless we improve our ways of programming with them [21]. Because many existing GPPLs support threads—and since this seems unlikely to change in the near future—we gear our efforts toward getting such improvements. In particular, our interest lies in solving problems with implementing *protocols among threads*. In this section, we identify three such problems.

At first sight, writing the computation code and the protocol code of a program using a single language may seem only natural. Indeed, many popular GPPLs have sufficient expressive power for doing so. Nevertheless, we consider it an inappropriate approach in many cases: typically, language designers gear GPPLs toward implementing computations. Implementing protocols, at a suitable level of abstraction, seems a secondary concern. Consequently, these languages work well for writing computation code, but not so for developing protocol code: the low-level concurrency constructs that they provide do not coincide with the higher-level concepts needed to express protocols directly. This results in two problems that complicate “writing code” for protocols.

Problem 1 (Implementing synchronization) *Threads communicating with each other using a shared memory, by directly reading from and writing to their common address space, must synchronize their actions. However, implementing synchronization using primitives such as locks, mutexes, semaphores, etc., comprises a tedious and error-prone activity.*

Problem 2 (Implementing coordination) *Threads interacting with each other in structured ways, according to some protocol, require coordination to ensure that they respect this protocol. However, implementing coordination using constructs such as assignments, **if**-statements, **while**-loops, etc., produces code that only indirectly conveys a protocol, which make it a tedious and error-prone activity.*

Interestingly, these two problems have a common cause: the lack of *appropriate abstractions* for implementing communication and interaction in GPPLs. For example: software developers should specify that a thread sends two integers and receives an array of rationals for a response—not that a thread allocates shared memory and performs pointer arithmetic. Or: developers should specify that communication between two threads inhibits interaction among other threads—not that threads acquire and release locks. Or: developers should specify that threads exchange data elements synchronously (i.e., atomically)—not that threads wait on a monitor until they get notified. We believe that programming languages should enable developers implementing communication and interaction to focus on the logic of the protocols

```

1 import java.util.LinkedList;
2 import java.util.concurrent.Semaphore;
3
4 public class Main {
5     private LinkedList<Object> buffer;
6     private Semaphore notEmpty;
7     private Semaphore notFull;
8
9     public Main() {
10         buffer = new LinkedList<Object>();
11         notEmpty = new Semaphore(0);
12         notFull = new Semaphore(1);
13         (new Producer()).start();
14         (new Producer()).start();
15         (new Consumer()).start();
16     }
17
18     private class Producer extends Thread {
19         public run() {
20             while (true) {
21                 Object d = produce();
22                 notFull.acquire();
23                 buffer.offer(d);
24                 notEmpty.release();
25             }
26         }
27     }
28
29     private class Consumer extends Thread {
30         public run() {
31             while (true) {
32                 notEmpty.acquire();
33                 Object d = buffer.poll();
34                 notFull.release();
35                 consume(d);
36             }
37         }
38     }
39 }

```

Figure 1: Java implementation of the producer–consumer example in [4, Algorithm 6.8].

involved—not on the realization of the necessary synchronization and coordination. We (should) have compilers for that.

In addition to the two problems identified above, the lack of appropriate abstractions in GPPLs causes a third problem: in the absence of proper structures to enforce (or at least encourage) good protocol programming practices, programmers frequently succumb to the temptation of not isolating protocol code. Conceptually, this problem differs from Problems 1 and 2, because it does not complicate “writing code” directly. However, it does perplex essentially everything else involved in a software development process. Although notions such as “modularization” [22] and “separation of concerns” [10] have long histories in computer science, linguistic support for their application in programming of concurrency protocols has scarcely received due attention. Modularization and separation of concerns have driven the development of modern programming languages and software development practices for decades. In fact, already in the early 1970s, Parnas attributed three advantages to abiding by these principles [22]:

“(1) managerial—development time should be shortened because separate groups would work on each module with little need for communication; (2) product flexibility—it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility—it should be possible to study the system one module at a time.”

Nevertheless, popular GPPLs do not enforce modularization of protocols. Consequently, dispersing protocol code among computation code comprises a common practice for implementing protocols among threads. To illustrate such dispersion—and its deficiencies—we discuss the producer–consumer Java code in Figure 1 (based on [4, Algorithm 6.8]). Two producer threads produce data elements and append them to a queue `buffer` (of size 1). Concurrently, a consumer thread takes elements from this queue and consumes them. While the queue `buffer` contains an element, the producers cannot append data until the consumer takes this element out of the queue. (We skip the methods `produce` and `consume`.) Easily, one can get the gist of the protocol involved in this example: the producers send—asynchronously, reliably, and in arbitrary order—data elements to the consumer. In contrast, one cannot easily point to coherent segments of the source code that actually implement this protocol. Indeed, only the combination of lines 5–7, 10–12, 21–23, and 28–30 does so. In this example, thus, we have not isolated the protocol in a distinct module; we have not separated our concerns. Therefore, the “Advantages of Modularization” identified by Parnas do not apply. In fact, the *monolithic program* in Figure 1 suffers from their opposites—the “Disadvantages of Dispersion:”

(I) Groups cannot work independently on computation code and protocol code of monolithic pro-

```

1 public interface Protocol {
2     public void offer(Object o);
3     public Object poll();
4 }
5 public class Main {
6     private Protocol protocol;
7     public Main() {
8         protocol = new P();
9         (new Producer()).start();
10        (new Producer()).start();
11        (new Consumer()).start();
12    }
13    private class Producer extends Thread {
14        public run() {
15            while (true) {
16                Object d = produce();
17                protocol.offer(d);
18            } } }
19    private class Consumer extends Thread {
20        public run() {
21            while (true) {
22                Object d = protocol.poll();
23                consume(d);
24            } } } }

```

Figure 2: Reimplementation of the producer–consumer program in Figure 1.

grams. Moreover, one cannot straightforwardly reuse computation code or protocol code of monolithic programs in other programs: this would require dissecting and disentangling the former.

- (2) Small changes to a protocol require nontrivial changes throughout a monolithic program. For instance, suppose that we allow the producers in the producer–consumer example to send data elements only in alternating order. Implementing such turn-taking requires significant changes.
- (3) One cannot study computation code and protocol code in isolation when they are entangled: to reason about protocol correctness, one must analyze monolithic programs in their entirety.

The impact of these shortcomings only increases when programs grow larger, interaction among threads intensifies, and protocol complexity increases—a likely situation in the current multicore era.

Problem 3 (Modularizing protocol code) *The lack of appropriate abstractions in GPLs tempts developers to disperse protocol code among code of computational tasks. In that case, developers do not isolate protocols in modules (e.g., classes, packages, namespaces), but intermix them with computations. This practice makes independently developing, maintaining, reusing, modifying, testing, and verifying protocol code problematic or impossible.*

To avoid the Disadvantages of Dispersion, we propose to isolate protocol code in separate modules. In object-oriented languages, for instance, one can achieve this by encapsulating all the protocol logic in a separate class. To illustrate this approach, we rewrote the monolithic program in Figure 1 as the *modular program* in Figure 2: we moved all the protocol code to a class P (see Section 4.2 for its implementation), which implements the interface `Protocol`.³ To such programs, the Advantages of Modularization apply. *First*, groups can develop protocol code (e.g., the implementations of the methods `offer` and `poll`) independently from computation code. Moreover, one can easily reuse protocol implementations. *Second*, changing the protocol requires changing only the class implementing the protocol (e.g., the class P); computation code, however, remains unaffected. *Third*, we can analyze the protocol in isolation by studying only the class implementing the protocol (e.g., the class P).

3 Solution: Protocol DSLs

In the previous section, we explained how the lack of appropriate abstractions complicates three aspects of implementing protocols: implementing synchronization (Problem 1), implementing coordina-

³The definition of the interface `Protocol` in Figure 2 serves only our present discussion: not every protocol has methods `offer` and `poll`. In general, the interface of a protocol should provide methods that computation code can invoke for executing this protocol. In the context of our present discussion, `offer` and `poll` seemed appropriate names.

tion (Problem 2), and modularizing protocol code (Problem 3). We believe that *domain-specific languages* (DSL) offer a solution to these problems.

Definition 1 (Domain-specific language [8]) *A domain-specific language is a programming language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.*

Domain-specific languages dedicated to the implementation of protocols solve Problems 1 and 2 *by this very definition*. Moreover, such *protocol DSLs* naturally force developers to isolate their protocols in modules: specifying protocols in a different language leads to a clear syntactic separation between computation code and protocol code. Consequently, using protocol DSLs in the following workflow secures the Advantages of Modularization.

- Developers write the computation code of an application in a GPPL.
- Developers specify protocols among threads in a protocol DSL.
- A DSL compiler compiles protocol specifications to GPPL code, seamlessly integrating protocols with computations.

While the benefits seem clear, one question remains: where to get these protocol DSLs from? Must we invent them from scratch? And if so, what kinds of “appropriate abstractions” should they provide?

Fortunately, we do not need to design everything from the ground up: interaction and concurrency have received plenty of attention from the theoretical computer science community over the past decades, and researchers have investigated high-level models of concurrency for many years (e.g., Petri nets, process algebras). This led to various formalisms for synchronizing and coordinating parties running concurrently (e.g., actors, agents, components, services, processes, etc.). We believe that these models of concurrency provide appropriate abstractions for specifying and reasoning about protocols (albeit, not all do so equally well). In other words, the protocol DSLs that we look for already exist. However, many of these formalisms lack sophisticated tool support, and in particular, the kind of compiler mentioned above. Therefore, we consider implementing such code generation tools among the main goals in our efforts toward alleviating the burden of programming with threads. But which existing concurrency formalism should we focus on?

4 Reo as a Protocol DSL

One model of concurrency has our particular interest: Reo [1, 2], an interaction-based model of concurrency with a graphical syntax, originally introduced for coordinating components in component-based systems. As with other models of concurrency, Reo has a solid foundation: there exist various compositional semantics [17] for describing the behavior of Reo programs, called *connectors*, along with tools for analyzing them. This includes both functional analysis (detecting deadlock, model-checking) and reasoning about nonfunctional properties (computing quality-of-service guarantees). Its declarative nature, however, distinguishes Reo from other models of concurrency. Using Reo, software developers specify *what*, *when*, *where*, and *why* interaction takes place; not *how*. Indeed, Reo does not feature primitive actions for sending or receiving data elements. Rather, Reo considers interaction protocols as constraints on such actions. In stark contrast to traditional models of concurrency, Reo’s constraint-based notion of interaction has the advantage that to formulate (specify, verify, etc.) protocols, one does not need to even consider any of the alternative sequences of actions that give rise to them.

Using Reo, computational threads remain completely oblivious to protocols that compose them into, and coordinate their interactions within, a concurrent application: their code contains no concurrency

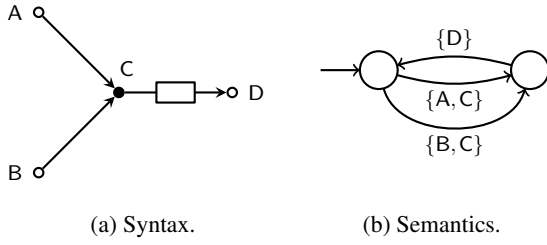


Figure 3: Syntax and semantics, as a port automaton, of MergerWithBuffer.

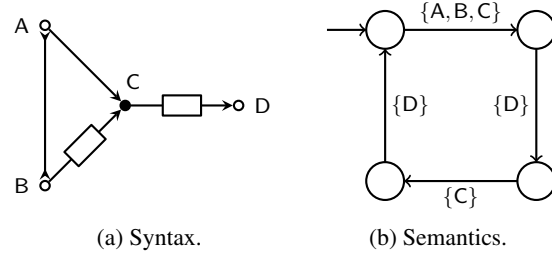


Figure 4: Syntax and semantics, as a port automaton, of AlternatorWithBuffer.

primitive (e.g., semaphore operations, signals, mutex, or even direct communication as in send/receive). The sole means of communication for a computational thread consists of I/O actions that it performs on its own input/output ports. To construct an application, one composes a set of such threads together with a protocol by identifying the input/output ports of the computational threads with the appropriate output/input nodes of a Reo connector that implements the protocol. A Reo compiler then generates the proper multithreaded application code.

We proceed as follows. In Section 4.1, we explain the main concepts of Reo through three example connectors, each of which represents a protocol that one can use in the producer–consumer example of Section 2. In Section 4.2, we discuss our Reo-to-Java compiler.

4.1 Reo by Example: Producer–Consumer Protocols

Figures 3a, 4a, and 5a show three example connectors (i.e., Reo programs): graphs of nodes and arcs, which we refer to as *channels*. We refer to nodes that admit I/O operations as *boundary nodes* and draw them as open circles in figures. Intuitively, one can interpret the graph representing a Reo connector as follows: data elements, dispatched on input (boundary) nodes by output operations, move along arcs to other nodes, which replicate them if they have multiple outgoing channels, along to output (boundary) nodes, from which input operations can fetch them. Groups of such (input, output, and transport) activities may take place atomically. Importantly, communicating parties that perform I/O operations on the boundary nodes of a connector remain oblivious to how the connector routes data: parties that fetch or dispatch data elements do not know where these elements come from or go to.

The connector in Figure 3a specifies the same protocol as the one embedded in the Java code in Figure 1. We can explain the behavior of this connector, named MergerWithBuffer, best by discussing the *port automaton* [20] that describes its semantics. Figure 3b shows this automaton (derived automatically from Figure 3a): every state corresponds to an internal configuration of MergerWithBuffer, while every transition describes a step of the protocol specified by MergerWithBuffer. Transitions carry a *synchronization constraint*: a set containing those nodes through which a data element passes in an atomic protocol step. Thus, in the initial state of MergerWithBuffer, a data element passes either nodes A and C or nodes B and C. Every element that passes C subsequently arrives at a buffer with capacity 1. We represent this buffer with a rectangle in Figure 3a. While the buffer remains full, no data elements can pass A, B, or C. In that case, the only admissible step results in the element stored in the buffer leave the buffer and pass through D.

Figure 4 shows another connector, named AlternatorWithBuffer, that one can use in the producer–consumer example. In contrast to MergerWithBuffer, AlternatorWithBuffer forces the producers to

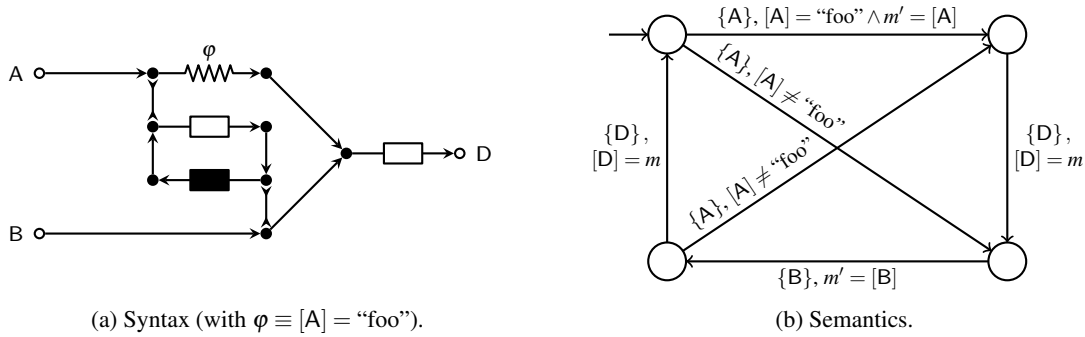


Figure 5: Syntax and semantics, as a constraint automaton, of $\text{SequencerWithBuffer}_\varphi$.

synchronize (represented by the arrow-tailed edge between nodes A and B): only if they can dispatch data elements simultaneously, the connector allows them to do so. In that case, the data element dispatched on A passes node C and enters the horizontal buffer; concurrently, the data element dispatched on B enters the diagonal buffer. In the next protocol step, the data element in the horizontal buffer leaves this buffer and passes node D. Subsequently, the data element in the diagonal buffer leaves this buffer, passes C, and enters the horizontal buffer. Finally, the data element now in the horizontal buffer (originally dispatched on B) leaves this buffer and passes D. Thus, $\text{AlternatorWithBuffer}$ first synchronizes the producers, and second, it offers their data elements in alternating order to the consumer.

Figure 5 shows a third connector, named $\text{SequencerWithBuffer}_\varphi$, that one can use in the producer-consumer example. The protocol specified by this connector differs in two significant ways from MergersWithBuffer and $\text{AlternatorWithBuffer}$. The first difference relates to (the lack of) synchronization: unlike $\text{AlternatorWithBuffer}$, $\text{SequencerWithBuffer}_\varphi$ does not force the producers to synchronize before they dispatch their data elements. (Similar to $\text{AlternatorWithBuffer}$, however, $\text{SequencerWithBuffer}_\varphi$ orders the sequence in which data elements arrive at the consumer.) The second difference relates to the *data-sensitivity* that $\text{SequencerWithBuffer}_\varphi$ exhibits: the zigzagged edge in Figure 5a represents a *filter channel* and we call φ a *filter constraint*: only those data elements satisfying its filter constraint propagate through a filter channel. In this example, we assume a simple filter constraint, namely $[A] = \text{"foo"}$, which means: the data element passing A equals the string “foo”.⁴ In other words, if a producer dispatches “foo” on A, the (right-horizontal) buffer becomes filled with “foo”; otherwise, the filter loses the dispatched data element, which means essentially, its producer has wasted its turn. In general, filter channels facilitate the specification of protocols whose execution depends on the content of the exchanged data.

Port automata cannot express the semantics of connectors with filter channels. For that, we need a stronger formalism: *constraint automata* (CA) [3], which support richer transition structures than port automata. Instead of only a synchronization constraint, transitions of CA carry also a *data constraint*: an expression about what the data passing particular nodes should look like in some protocol step. Figure 5b shows the constraint automaton that describes the semantics of $\text{SequencerWithBuffer}_\varphi$ (we omitted its nonboundary nodes from this CA). The symbols m and m' refer to the content of the (right-horizontal) buffer while and after a transition fires: $m' = [A]$ means that this buffer contains the value exchanged through A after a transition; $[D] = m$ means that the content of this buffer passes through D during a transition.

⁴Alternatively, one can formulate filter constraints as regular expressions or *patterns*. See [1].

4.2 Compiling Reo to Java

Next, we discuss how to use Reo as a DSL for implementing protocols: we present here an early version of our Reo-to-Java compiler, because it is simpler to explain. Although we focus on Java here, we emphasize the generality of our approach: nothing in Reo prevents us from compiling Reo to Fortran, C, C++, Objective-C, C#, or Visual Basic.

Before we can compile anything, we need a means to implement the “paper-and-pencil drawings” of connectors. We use the Reo IDE for this purpose, called *the Extensible Coordination Tools* (ECT).⁵ The ECT consists of a collection of Eclipse plug-ins, including a drag-and-drop editor for drawing connector diagrams. Under the hood, the ECT stores and manipulates such diagrams as XML documents. These XML documents serve as input to our Reo-to-Java compiler, detailed next.

Previously, we introduced Reo in terms of how data elements move through a connector, not unlike dataflow programming. Compiling connectors to some kind of distributed application, therefore, may seem an obvious choice: nodes naturally map to processing elements (e.g., cores), and the connections between processing elements can serve as channels. However, this approach has several shortcomings. *First*, the network topology of the hardware may not correspond with the topology of the connector that we want to deploy. *Second*, emulating Reo channels with hardware connections requires additional computations from the processing elements connected. This destroys the original idea of mapping Reo channels to hardware connections. *Third*, achieving the global atomicity, synchronization, and exclusion emerging in a connector requires complex distributed algorithms. Such algorithms inflict communication and processing overhead, deteriorating performance. In short, construing connectors and their topology too literally seems a bad idea in the context of compilation. Instead, our Reo-to-Java compiler compiles connectors based on their constraint automaton (CA) semantics.

The ECT ships with the CA of many common channels, including those in Figures 3, 4, and 5. By combining such primitive CA, through the act of composition [3], the ECT can automatically compute the CA of larger connectors. We use this open source CA library in our Reo-to-Java compiler: on input of an XML document specifying a connector, our compiler first computes the corresponding CA. Subsequently, it annotates this CA with Java identifiers. Finally, it produces a Java class using ANTLR’s StringTemplate technology [23]. One can use the resulting class as any Java class. By using CA for compiling connectors, we conveniently abstract away their nonboundary nodes.

To illustrate the compilation process, suppose that we want to compile `MergerWithBuffer` for use in the producer–consumer example of Section 2. After drawing `MergerWithBuffer` using the ECT, we feed the corresponding XML document to our Reo-to-Java compiler. This tool automatically generates a Java class `MergerWithBuffer` based on the CA semantics of `MergerWithBuffer`. More precisely, `MergerWithBuffer` objects run state machines representing the protocol specified by `MergerWithBuffer`. Figure 6 shows (parts of) the Java class generated by compiling `MergerWithBuffer`. We discuss some of its salient aspects.

- The class `MergerWithBuffer` extends the class `Thread` (line 1). By running connectors in their own thread, we enable them to proactively sense their environment for I/O operations; with massive-scale concurrent hardware, ample cores to run connectors on should always exist.
- Instances of `MergerWithBuffer` listen to three *ports* (line 7), which grant “computation threads” access to the boundary nodes of `MergerWithBuffer`. All interaction between computation threads and a `MergerWithBuffer` object occurs through the latter’s ports: computation threads can perform I/O operations—writes and takes—on ports, which in turn suspend threads until their op-

⁵ <http://reo.project.cwi.nl/>


```

1 public class MergerWithBuffer extends Thread {
2     /* The current state. */
3     private State current = State.EMPTY;
4     private enum State { FULL, EMPTY }
5
6     /* The boundary nodes of this connector. */
7     private Port A; private Port B; private Port D;
8
9     /* The data constraints this connector checks and */
10    /* the memory cells this connector has access to. */
11    // ---snip---
12
13    /* A random number generator for selecting transitions. */
14    Random random = new Random();
15
16    /* Constructs a MergerWithBuffer. */
17    public MergerWithBuffer(Port A, Port B, Port D) {
18        this.A = A; this.B = B; this.D = D;
19
20        /* Initialize data constraints. */
21        // ---snip---
22    }
23
24    /* Runs the state machine modeling MergerWithBuffer. */
25    public void run() {
26        while (true) {
27            switch (current) {
28                case State.EMPTY:
29                    switch (random.nextInt(2)) {
30                        case 0: tFromEmptyToFullA(); break;
31                        case 1: tFromEmptyToFullB(); break;
32                    }
33                    break;
34                case State.FULL:
35                    // ---snip---
36                    break;
37            } } }
38
39    /* Makes a transition from state EMPTY to state FULL, firing A. */
40    private void tFromEmptyToFullA() {
41        /* Lock and get pending writes. */
42        Set<Write> writesOnA = A.lockAndGetWrites();
43        if (writesOnA.isEmpty()) { abort(); return; }
44
45        /* Lock and get pending takes. */
46        Set<Take> takesOnD = A.lockAndGetTakes();
47        if (takesOnD.isEmpty()) { abort(); return; }
48
49        /* Check the synchronization and data constraints. */
50        if (/* ---snip--- */) {
51
52            /* Process writes and takes. */
53            A.performAndUnlock(/* ---snip--- */);
54            D.performAndUnlock(/* ---snip--- */);
55
56            /* Update memory cells. */
57            // ---snip---;
58
59            /* Update state. */
60            current = State.FULL;
61        }
62        abort();
63    }
64
65    /* Makes a transition from state EMPTY to state FULL, firing B. */
66    private void tFromEmptyToFullB() {
67        // ---snip---
68    }
69
70    /* Aborts a transition by unlocking all that it may have locked. */
71    private void abort() {
72        A.unlockWrites(); B.unlockWrites();
73        D.unlockTakes();
74    } }

```

Figure 6: (Parts of the) Java class generated by compiling MergerWithBuffer (see also Figure 3).

erations succeed. More technically, ports extend concurrent data structures called *synchronization points*:⁶ pairs of sets—one containing pending writes, another containing pending takes—supporting and admissible to *two-phase locking* schemes (see below) [5].

The class SyncPoint exposes the following methods:

- lockAndGetWrites() locks and returns the set of pending write operations (line 42).
- lockAndGetTakes() locks and returns the set of pending take operations (line 46).
- unlockWrites() and unlockTakes() unlock the sets of writes and takes (lines 72–73).
- performAndUnlock(Write) performs the specified write operation (first parameter) and unlocks the set of pending write operations (line 53).
- performAndUnlock(Take, Object) performs the specified take operation (first parameter) by passing it the data element to take (second parameter) and unlocks the set of pending take operations (line 54).
- The overridden method run() implements a state machine corresponding to the input CA of the compilation process (lines 25–37). The main loop never terminates (line 26). In each iteration,

⁶Synchronization points resemble π -calculus channels.

```

1 public class P implements Protocol {
2     private Port A = new Port();
3     private Port B = new Port();
4     private Port D = new Port();
5     private Map<Thread, Port> threads =
6         new ConcurrentHashMap<Thread, Port>();
7
8     public P() {
9         new MergerWithBuffer(A, B, D).start();
10    }
11    public Object poll() { return D.take(); }
12    public void offer(Object o) {
13        Thread thread = Thread.currentThread();
14        if (!threads.containsKey(thread))
15            synchronized (this) {
16                threads.put(thread,
17                    !threads.containsValue(A) ? A : B);
18            }
19        threads.get(thread).write(o);
20    } }

```

Figure 7: Class P.

it randomly selects a transition (line 29) going out of the current state (line 3). We collect code responsible for making transitions in separate methods (lines 39–63 and 66–68).

- An important step in the process of making a transition consists of checking its synchronization and data constraints (line 50). To do this in a thread-safe manner, a `MergerWithBuffer` object employs a two-phase locking scheme. During the *growing phase*, it acquires the locks of:
 - the set of pending writes of each port providing access to an input node (line 42);
 - the set of pending takes of each port providing access to an output node (line 46).

A `MergerWithBuffer` object locks only the sets of those boundary nodes that actually occur in the constraints under investigation. Later, during the *shrinking phase*, it releases these locks again (lines 43, 47, and 62). Only between phases, a `MergerWithBuffer` object checks the constraints under investigation. If they hold, it fires the corresponding transition, transporting data elements accordingly and removing the operations involved from the sets it has locked (lines 53–54).

To use the class `MergerWithBuffer` in the producer–consumer example of Section 2, we should incorporate it in the implementation of the class `P`, encountered before on line 8 in Figure 2; Figure 7 shows this implementation. Line 19 specifies that a producer performs a (blocking) write operation on the port assigned to it; line 11 specifies that a consumer performs a (blocking) take operation. The rest of `P` consists of initialization code. The latter characterizes the use of Reo as a protocol DSL: implementations of the `Protocol` interface serve as wrappers for compiled connectors, encapsulating all the protocol logic.

To change Figure 1 such that it respects the protocol specified by `AlternatorWithBuffer` requires nontrivial modifications across the source code. In contrast, we can straightforwardly implement a class `Q` **implements** `Protocol` and replace `P()` with `Q()` on line 8 in Figure 2. In fact, `Q` would differ from `P` only on line 9 in Figure 7: in `Q`, we would construct an `AlternatorWithBuffer` object instead of a `MergerWithBuffer` object. Similarly, we can use the protocol specified by `SequencerWithBufferφ`. This shows that using Reo, we can easily change protocols without affecting computation code.⁷

This subsection demonstrates the feasibility of modularizing protocols and implementing protocol DSLs. We remark that this approach does not preclude the use of *dedicated implementations* for certain parts of a protocol. For instance, consider the buffer of `MergerWithBuffer`. Our Reo-to-Java compiler implements this buffer using shared memory and explicit locks (transparent to software developers using Reo, though). But suppose that the architecture we deploy our producer–consumer program on features also *hardware transactional memory* (HTM) [12]. Our approach allows one to write a dedicated

⁷More precisely, handwritten computation code and protocol code generated by our Reo compiler communicate only through shared ports; these ports do not change when replacing one connector with another. Thus, unless the number of ports changes, a syntactically valid program remains syntactically valid.

implementation of buffers that exploits this HTM. Subsequently, we can replace the standard buffer implementation with this dedicated implementation.⁸ Thus, besides high-level constructs by default, our approach offers developers the flexibility of applying lower-level languages if necessary.

5 Concluding Remarks

Our current work focuses on improving our Reo-to-Java compiler. For instance, the classes currently generated by our compiler execute sequentially. We can parallelize this rather straightforwardly by checking ports for appropriate I/O operations concurrently for different transitions. However, we speculate that we can get even better performance if, instead, we optimize at the semantic level: we wish to decompose automata into “smaller” automata that can execute concurrently without synchronization while preserving the original semantics (see [18] for preliminary results). Another potential optimization involves scheduling: the formal semantics of connectors provide very tangible information for scheduling threads. Exploiting this information should yield substantial performance gains. Hopefully, such improvements make our approach a competitive alternative to lower level approaches also in terms of performance.

In recent years, *session types* [14, 15] have entered the realm of object-oriented programming (recent work includes [6, 9, 11, 13, 16]). Although session types comprise a valuable new technique for programming with threads, we wonder if the abstractions provided by them suffice. Still, we consider it a very interesting development, especially since Reo does not feature types; extending Reo with session types would comprise a significant improvement.

Finally, although we focused on implementing protocols among threads in this paper, the Reo-to-Java compiler presented has proven itself useful also in the domain of Web Service orchestration [19].

References

- [1] Farhad Arbab (2004): *Reo: a channel-based coordination model for component composition*. *MSCS* 14(3), pp. 329–366, doi:10.1017/S0960129504004153.
- [2] Farhad Arbab (2011): *Puff, The Magic Protocol*. In Gul Agha, Olivier Danvy & José Meseguer, editors: *Formal Modeling: Actors, Open Systems, Biological Systems, LNCS 7000*, Springer, pp. 169–206, doi:10.1007/978-3-642-24933-4_9.
- [3] Christel Baier, Marjan Sirjani, Farhad Arbab & Jan Rutten (2006): *Modeling component connectors in Reo by constraint automata*. *SCICO* 61(2), pp. 75–113, doi:10.1016/j.scico.2005.10.008.
- [4] Mordechai Ben-Ari (2006): *Semaphores*. In: *Principles of Concurrent and Distributed Programming*, 2nd edition, Addison-Wesley, pp. 107–144.
- [5] Philip Bernstein, Vassos Hadzilacos & Nathan Goodman (1987): *Two Phase Locking*. In: *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, pp. 47–111.

⁸Roughly, first, we write an HTM-based implementation of buffers. Second, we remove the connection between nodes C and D in Figure 4a. Third, we generate code for the subconnector containing nodes A, B, and C and for the subconnector containing node D. (The latter consists of only node D. However, by the semantics of Reo, we can replace this by an equivalent connector of two synchronizing nodes.) Thus, we now have two compiled connectors. Finally, we place the HTM-based implementation of buffers between these two compiled connectors, as an active entity: our dedicated buffer implementation, which performs writes and takes on C and D, runs in its own thread—effectively, it operates as a fourth party involved in the protocol. Interestingly, the “real” communicating parties remain oblivious to the introduction of this fourth party. Further optimization can eliminate the active entity that performs writes and takes on C and D, merging its functionality in the behavior of nodes C and D.

- [6] Sara Capecchi, Mario Coppo, Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou & Elena Giachino (2009): *Amalgamating sessions and methods in object-oriented languages with generics*. TCS 410(2–3), pp. 142–167, doi:10.1016/j.tcs.2008.09.016.
- [7] Sung-Eun Choi & Christopher Lewis (2000): *A Study of Common Pitfalls in Simple Multi-Threaded Programs*. In: *Proceedings of SIGCSE*, pp. 325–329, doi:10.1145/330908.331879.
- [8] Arie Van Deursen, Paul Klint & Joost Visser (2000): *Domain-Specific Languages: An Annotated Bibliography*. ACM SIGPLAN Notices 35(6), pp. 26–36, doi:10.1145/352029.352035.
- [9] Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous & Nobuko Yoshida (2009): *Objects and session types*. IC 207(5), pp. 595–641, doi:10.1016/j.ic.2008.03.028.
- [10] Edsger Dijkstra (1982): *On the Role of Scientific Thought (EWD447)*. In: *Selected Writings on Computing: A Personal Perspective*, Texts and Monographs in Computer Science, Springer, pp. 60–66, doi:10.1007/978-1-4612-5695-3_12.
- [11] Simon Gay, Vasco Vasconcelos, António Ravara, Nils Gesbert & Alexandre Caldeira (2010): *Modular Session Types for Distributed Object-Oriented Programming*. In: *Proceedings of POPL 2010*, pp. 299–312, doi:10.1145/1706299.1706335.
- [12] Maurice Herlihy & Eliot Moss (1993): *Transactional Memory: Architectural Support for Lock-Free Data Structures*. ACM SIGARCH Computer Architecture News 21(2), pp. 289–300, doi:10.1145/173682.165164.
- [13] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen & Nobuko Yoshida (2011): *Scribbling Interactions with a Formal Foundation*. In Raja Natarajan & Adegboyega Ojo, editors: *Distributed Computing and Internet Technology*, LNCS 6536, Springer, pp. 55–75, doi:10.1007/978-3-642-19056-8_4.
- [14] Kohei Honda, Vasco Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In Chris Hankin, editor: *Programming Languages and Systems*, LNCS 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567.
- [15] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty Asynchronous Session Types*. In: *Proceedings of POPL 2008*, pp. 273–284, doi:10.1145/1328438.1328472.
- [16] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida & Kohei Honda (2010): *Type-Safe Eventful Sessions in Java*. In Chris Hankin, editor: *ECOOP 2010 — Object-Oriented Programming*, LNCS 6183, Springer, pp. 329–353, doi:10.1007/978-3-642-14107-2_16.
- [17] Sung-Shik Jongmans & Farhad Arbab (2012): *Overview of Thirty Semantic Formalisms for Reo*. SACS 22(1), pp. 201–251, doi:10.7561/SACS.2012.1.201.
- [18] Sung-Shik Jongmans, Dave Clarke & José Proença (2012): *A Procedure for Splitting Processes and its Application to Coordination*. EPTCS 91(1), pp. 79–96, doi:10.4204/EPTCS.91.6.
- [19] Sung-Shik Jongmans, Francesco Santini, Mahdi Sargolzaei, Farhad Arbab & Hamideh Afsarmanesh (2011): *Automatic Code Generation for the Orchestration of Web Services with Reo*. In Flavio De Paoli, Ernesto Pimentel & Gianluigi Zavattaro, editors: *Service-Oriented and Cloud Computing*, LNCS 7592, Springer, pp. 1–16, doi:10.1007/978-3-642-33427-6_1.
- [20] Christian Koehler & Dave Clarke (2009): *Decomposing Port Automata*. In: *Proceedings of SAC 2009*, pp. 1369–1373, doi:10.1145/1529282.1529587.
- [21] Edward Lee (2006): *The Problem with Threads*. Computer 39(5), pp. 33–42, doi:10.1109/MC.2006.180.
- [22] David Parnas (1972): *On the Criteria To Be Used in Decomposing Systems into Modules*. CACM 15(12), pp. 1053–1058, doi:10.1145/361598.361623.
- [23] Terence Parr (2007): *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf.